Le Remote Core Lock (RCL) : une nouvelle technique de verrouillage pour les architectures multi-coeur

Jean-Pierre Lozi

RenPar'20 / SympA'14 / **CFSE 8**
Saint-Malo, France, du 10 au 13 mai 2011

Le Remote Core Lock (RCL) : une nouvelle technique de verrouillage pour les architectures multi-cœur

Jean-Pierre Lozi
Laboratoire d'Informatique de Paris 6 (LIP6)

Résumé

Les architectures multi-cœur sont désormais omniprésentes dans les systèmes informatiques personnels et d'entreprise. À l'heure actuelle, les systèmes et les applications sont cependant incapables d'exploiter efficacement la puissance de ces nouvelles architectures, en particulier à cause du coût d'exécution des sections critiques.

Nous proposons une nouvelle approche, baptisée Remote Core Lock (RCL), qui permet d'améliorer les performances des applications multi-fil sur les architectures multi-cœur. Le principe du RCL est de remplacer, dans les applications patrimoniales, certaines prises de verrous critiques en terme de performances par des appels de procédures distantes sur un coeur dédié appelé serveur. L'intérêt du RCL est double. D'une part, en remplaçant les demandes de prises de verrou par un unique envoi de message au serveur, le RCL évite les effets d'effondrement liés à la surcharge du bus lors d'un grand nombre de demandes concurrentes de prise de verrou. D'autre part, les verrous sont en général utilisés pour protéger les accès à des données partagées et le RCL évite la migration de ces données sur le coeur qui prend le verrou : les données partagées restent en effet dans les caches du serveur, puisque celui-ci est le seul à y accéder. Nos premières évaluations montrent que (i) le RCL offre des performances supérieures aux verrous classiques en cas de forte contention sur le bus, (ii) grâce au RCL, le benchmark SPLASH-2/Raytrace passe à l'échelle jusqu'à 32 cœurs, au lieu de 8 avec des verrous classiques et (iii) l'utilisation du RCL dans le serveur de cache memcached offre un gain de débit allant jusqu'à 65%.

1. Introduction

Les architectures multi-cœur sont désormais omniprésentes dans les systèmes informatiques, des serveurs d'entreprise aux ordinateurs de bureaux, en passant par les systèmes embarqués. La technologie NUMA (Non-Uniform Memory Access) permet l'utilisation d'un très grand nombre de cœurs au coût d'une hiérarchie complexe de caches et de bus mémoire. Néanmoins, les systèmes et applications actuels ne permettent pas d'exploiter au maximum le potentiel de ces nouvelles architectures : les performances des applications stagnent malgré la forte augmentation du nombre de cœurs.

L'obstacle majeur auxquels se heurtent les développeurs lorsqu'il s'agit de faire passer à l'échelle les applications multi-fil classiques sur les architectures à grand nombre de cœurs est le coût d'exécution de blocs de code en exclusion mutuelle. En effet, les sections critiques, protégées par des verrous dans la majorité des cas, sont souvent la cause de goulots d'étranglement, et ce pour deux raisons principales. D'une part, le coût d'une prise de verrou peut être prohibitif lorsque de nombreux processus essaient d'entrer en section critique en même temps, à cause de la forte contention sur le bus mémoire. D'autre part, lors de l'exécution d'une section critique, le temps d'accès à des données se trouvant dans des caches ou des bancs mémoires éloignés du cœur local peut être important à cause de la migration des données sous-jacente. Ces problèmes sont cruciaux sur les architectures actuelles où la communication entre un nombre toujours croissant de cœurs d'exécution est assurée par des algorithmes de cohérence de cache de plus en plus coûteux.

Nous proposons une solution baptisée le Remote Core Lock (RCL) pour améliorer les performances des applications patrimoniales multi-fil en mémoire partagée. Le principe du RCL est de remplacer des prises de verrou par des appels de procédures distantes sur un coeur dédié appellé serveur. Le gain espéré est double. D'une part, en remplaçant les demandes de prises de verrou par un transfert de contrôle rapide basé sur un envoi unique d'une ligne de cache du client au serveur, la latence d'entrée en section critique diminue significativement. En effet, ce mécanisme évite la concurrence d'accès sur une variable unique modélisant un verrou et limite ainsi la contention sur le bus de données. D'autre part, le RCL évite la migration des données partagées auxquelles accède la section critique vers le coeur qui prend le verrou comme c'est le cas avec des verrous traditionnels. En effet, comme toutes les sections critiques

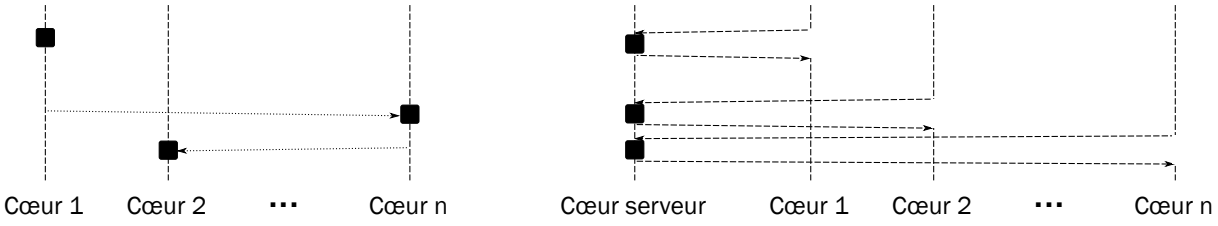


FIGURE 1 – Fonctionnement des verrous classiques et du RCL. Avec le RCL, toutes les sections critiques (carrés noirs) sont exécutées sur le cœur serveur, qui possède les données partagées auxquelles elles accèdent. Ces données n'ont donc pas à migrer d'un cœur à l'autre comme c'est le cas avec des verrous classiques (flèches sur l'illustration de gauche).

sont exécutées sur le cœur serveur, les données partagées auxquelles elles accèdent sont toujours présentes dans ses caches ou dans ses bancs mémoire locaux. Comparé aux travaux qui visent à améliorer les performances de la prise de verrou [5, 19, 29, 12, 18, 26], le RCL permet en plus d'améliorer la localité des données à l'intérieur de la section critique, et comparé aux travaux qui visent à proposer de nouveaux paradigmes de programmation pour gérer les architectures multicoeurs [9, 10, 13, 6], le RCL évite une refonte complète du code puisqu'il ne s'agit que de transformer les sections critiques en appels de procédure distantes, mécanisme simple et automatisable.

Les travaux de mise au point et d'évaluation du RCL sont en cours de réalisation. Nos contributions sont les suivantes :

- Une implémentation efficace du RCL. En faisant explicitement intervenir la notion de lignes de cache, notre implémentation permet de limiter au maximum la contention sur le bus de données et le nombre de défauts de cache. Cette gestion optimisée de la mémoire permet de limiter (i) le coût d'entrée en section critique, même lorsque la contention sur le bus de données est forte, et (ii) la durée d'exécution des sections critiques, grâce à une amélioration de la localité des données.
- Une méthodologie permettant de remplacer un verrou par un RCL dans les applications patrimoniales. En effet, par sa nature, le RCL ne peut être utilisé comme un verrou classique, c'est-à-dire avec deux simples fonctions de verrouillage. Il est nécessaire d'isoler (i) le code de la section critique qui est encapsulé dans une fonction dont un pointeur est passé au cœur serveur lorsque son exécution est demandée, et (ii) le contexte d'exécution de la section critique concernée, c'est-à-dire les variables privées utilisées en entrée et en sortie, afin de les envoyer au cœur serveur. Une description plus poussée de ce procédé est donnée en section 2.
- Une première série d'évaluations du RCL. Nous montrons d'abord, à l'aide d'un microbenchmark, que le RCL est plus efficace que les verrous classiques lorsque la contention est forte, même si le nombre de données partagées auxquelles les sections critiques accèdent est très limité. Un accès à un nombre réduit de variables partagées dans les sections critiques (10 lignes de cache dans la plupart des cas) suffit a rendre le RCL plus efficace que les approches classiques quelle que soit le degré contention. L'utilisation du RCL dans le benchmark patrimonial Raytrace de la suite SPLASH-2 entraîne des gains de performance et de scalabilité importants : dans le cas de la scène « car », par exemple, on observe un gain de performances de plus de 70%, et un passage à l'échelle jusqu'à au 32 cœurs contre 8 cœurs avec des verrous classiques. Similairement, l'utilisation du RCL dans le serveur de cache memcached offre un gain de débit de plus de 65%.

Dans la section 2, nous détaillerons le fonctionnement et la mise en œuvre du RCL dans les applications patrimoniales. Dans la section 3, nous présenterons nos premiers résultats d'évaluation du RCL. Dans la section 4, nous présenterons d'autres travaux qui se sont concentrés sur l'amélioration des performances des applications sur les architectures multi-cœur. Enfin, nous conclurons dans la section 5.

2. Mise en œuvre du RCL

Nous proposons une nouvelle technique de verrouillage nommée le Remote Core Lock (RCL) dont la caractéristique principale est la délégation de l'exécution des sections critiques à un cœur dédié. Son fonctionnement est illustré par la Figure 1. Le fil de contrôle s'exécutant sur le cœur dédié, appellé

serveur, attend en permanence des demandes d'exécution de sections critiques des fils de contrôle de l'application originale, appellés clients. L'exécution d'un RCL est similaire à celle d'un Remote Procedure Call (RPC) optimisé, le code du RPC étant celui de la section critique. Le serveur n'a pas besoin de prendre de verrou pour traiter les demandes d'exécution de sections critiques provenant des clients, car il est le seul à les exécuter : leur traitement est donc naturellement sérialisé. De plus, comme le cœur serveur est le seul à exécuter les sections critiques, il est également le seul à accéder à leurs données partagées. Ces données sont donc très souvent présentes dans les caches et bancs mémoires proches du cœur serveur, d'où un nombre réduit de défauts de cache lors de l'exécution des sections critiques.

Notons qu'un programme peut utiliser plusieurs verrous. Si les sections critiques ne sont chacune protégées que par une unique prise de verrou, cela revient à dire qu'on a n ensembles $\mathcal{E}_i$ ($0 \leq i < n$) de sections critiques, tels que deux sections critiques de l'un de ces ensembles $\mathcal{E}_i$ ne doivent jamais être exécutées en parallèle, mais deux sections critiques provenant d'ensembles différents $\mathcal{E}_i$ et $\mathcal{E}_j$ ($i \neq j$) peuvent être exécutées en parallèle. Dans ce cas, pour obtenir des performances optimales au prix d'une forte consommation de ressources (i.e., de cœurs d'exécution), il est possible d'utiliser un cœur serveur par ensemble $\mathcal{E}_i$: cela suffira pour assurer la sérialisation au sein de ces ensembles. Pour limiter le nombre de cœurs serveurs, on pourra affecter le traitement de plusieurs ensembles $\mathcal{E}_i$ au même serveur. Afin de simplifier la présentation, dans la suite de cette section, le RCL ne sera utilisé que pour remplacer un seul verrou, mais l'utilisation du RCL pour plusieurs verrous s'en déduit simplement, du moment que les sections critiques concernées ne sont chacunes protégées que par une seule prise de verrou : il suffit pour les clients de toujours demander l'exécution des sections critiques correspondant à un verrou donné au même serveur. L'utilisation du RCL s'avère plus complexe lorsque les sections critiques concernées nécessitent des prises de verrou multiples. Cette problématique ne sera cependant pas traitée dans cet article : ce sera le sujet d'une étude ultérieure.

2.1. Contraintes sur le protocole

Le protocole de synchronisation entre le serveur et les clients est soumis à deux contraintes principales. D'une part, le serveur doit être générique. En effet, pour des raisons de praticité, il n'est pas souhaitable qu'il soit nécessaire d'implémenter un serveur spécifique pour chaque verrou : le serveur doit donc être capable d'exécuter n'importe quelle section critique. D'autre part, le protocole doit assurer une réactivité importante : pour qu'il soit plus rentable d'exécuter les sections critiques sur un cœur distant que sur un cœur local, il est nécessaire que les transferts de contrôle d'un cœur à un autre soient très rapides.

Afin que la contrainte de généricité soit assurée, le code des sections critiques isolées par le verrou qu'on souhaite convertir en RCL doit être isolé et encapsulé dans des fonctions. De cette manière, pour exécuter une section critique, le client envoie le pointeur vers la fonction encapsulant le code correspondant à cette section critique au serveur. Le serveur est ainsi indépendant des sections critiques qu'il exécute : son code peut être intégré à une bibliothèque et son fonctionnement peut être ignoré des développeurs. De plus, l'utilisation de fonctions encapsulant le code des sections critique a l'avantage de simplifier la sémantique du transfert des variables privées nécessaires à leur exécution. En effet, en plus des variables partagées, les sections critiques accèdent parfois aussi à des variables privées au fil de contrôle client, et ce, en lecture ou en écriture. On appelle ces variables le contexte de la section critique. Grâce à l'utilisation de pointeurs de fonctions, transférer le contexte de la section critique au serveur est simplifié : il suffit de l'encapsuler dans une structure, et de passer un pointeur vers cette structure au serveur lors du transfert de contrôle.

Afin de satisfaire la contrainte de réactivité, nous utilisons exclusivement des mécanismes d'attente active pour la synchronisation entre le serveur et les clients, au lieu de mécanismes bloquants qui nécessitent l'endormissement des fils de contrôle. Si l'attente active est la technique la plus efficace pour obtenir une bonne réactivité des mécanismes de synchronisation, elle a deux désavantages principaux. Le premier désavantage est la forte consommation de temps CPU qu'elle engendre. Ce coût est cependant à relativiser dans l'optique d'une architecture disposant d'un très grand nombre de cœurs et donc de ressources CPU importantes. Le second désavantage est l'effondrement des performances qui se produit lorsque de nombreux fils de contrôle essaient de modifier une variable de synchronisation partagée

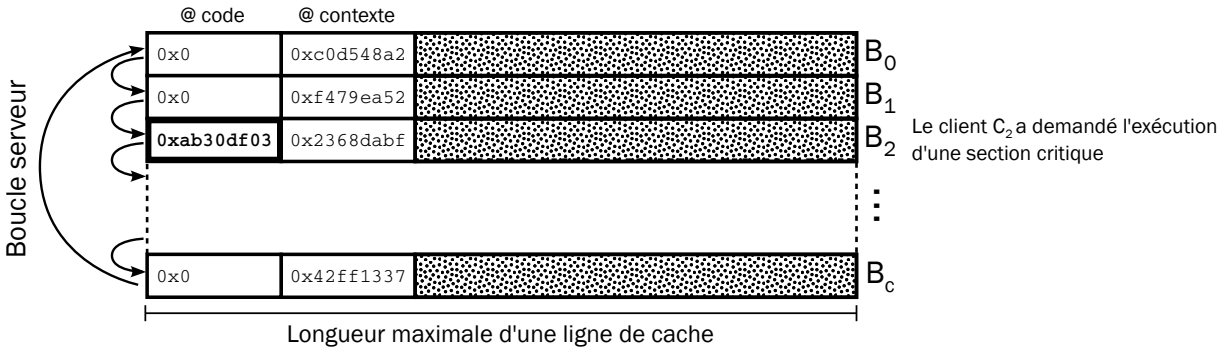


FIGURE 2 – Bloc de mémoire `sync`. Celui-ci est divisé en blocs de la taille maximale d'une ligne de cache au niveau du matériel. Le serveur attend activement que le premier mot de l'un de ces blocs contienne une adresse : lorsque c'est le cas, un client désire exécuter un RPC.

en même temps avec une instruction atomique de type compare-and-swap [1], à cause de la surcharge engendrée sur le bus de donnée. C'est là le défaut principal des verrous à attente active classiques. Pour résoudre ce problème, chaque client communique avec le serveur à travers une variable de synchronisation dédiée. Ce mécanisme est rendu possible par la structure centralisée du RCL, et lui confère une forte résistance aux phénomènes d'écroulement lorsque de nombreux clients doivent exécuter des sections critiques en même temps.

2.2. Mécanisme

Le mécanisme de communication entre le serveur et les clients du RCL est illustré par la Figure 2. Le serveur et les clients partagent un espace mémoire nommé `sync` de taille $c \cdot l$ où c est le nombre de clients et l la taille de la plus grande ligne de cache au niveau du matériel. Cet espace mémoire est divisé en c blocs de données B_i ($0 \leq i < c$) de taille l, de manière à ce que chaque client C_i ($0 \leq i < c$) puisse communiquer avec le serveur en écrivant dans le bloc B_i. Pour améliorer les performances, la structure `sync` est alignée de façons à ce que chaque bloc B_i occupe complètement une ligne du cache de plus haut niveau du point de vue du matériel : cela permet d'éviter tout faux partage. Les deux premiers mots de 64 bits de chaque bloc B_i sont respectivement destinés à contenir (i) l'adresse de la méthode contenant le code de la section critique à exécuter pour le client C_i ou 0 si ce client n'a pas besoin d'exécuter de section critique, et (ii) l'adresse de la structure encapsulant le contexte, c'est-à-dire les variables accédées en lecture (variables d'entrée) et celles accédées en écriture (variables de sortie) par la section critique.

Serveur

Le fil de contrôle serveur itère sur les blocs B_i ($0 \leq i < c$) en attendant que l'un des blocs contienne une valeur différente de zéro dans son premier mot. Lorsque c'est le cas pour le bloc B_i, cela signifie que le client C_i désire exécuter une section critique : le serveur l'exécute donc, grâce à l'adresse de la fonction qu'il récupère dans le premier mot et à l'adresse du contexte qu'il récupère dans le second mot. Lorsque le traitement de la section critique est terminé, le serveur écrit 0 dans le premier mot du bloc B_i pour prévenir le client C_i que sa section critique a pu être exécutée, et il recommence à itérer sur les blocs B_i en attendant une autre demande d'exécution d'une section critique.

Client

Du point de vue du client C_i, pour exécuter une section critique, il s'agit de (i) remplir le second mot du bloc B_i avec l'adresse de la structure encapsulant le contexte de la section critique puis de (ii) remplir le premier mot du bloc B_i avec l'adresse de la fonction correspondant à la section critique souhaitée. Le client attend ensuite que le premier mot du bloc B_i passe à 0 (attente active), ce qui signifie pour lui que le serveur a fini d'exécuter sa section critique.

1. L'instruction compare-and-swap (CAS) compare le contenu d'une adresse mémoire avec une valeur donnée et, si les valeurs comparées sont les mêmes, remplace le contenu de l'adresse mémoire par une autre valeur fournie en paramètre. Il s'agit d'une unique instruction processeur, et donc d'une opération atomique.

```
1  int func(int p1, int p2)
2  {
3      int v1, v2;
4      int r1, r2;
5
6      v1 = ...;
7
8      LOCK();
9
10     /* Critical section */
11     ...;
12     ... = p1;
13     ... = v1;
14     r1 = ...;
15     ...;
16
17     UNLOCK();
18
19     r2 = ...;
20
21     return r1 + r2;
22 }
```

Figure 3.1. Exemple de section critique isolée par des fonctions de prise et de relâche de verrou classique.

```
1  typedef struct func_cs_ctx {
2      /* Input parameters */
3      int p1;
4      int v1;
5      /* Output parameters */
6      int r1;
7  } func_cs_ctx_t;
8
9  void func_cs(void *_ctx)
10 {
11     func_cs_ctx_t *ctx = (func_cs_ctx_t *)_ctx;
12
13     /* Critical section */
14     ...;
15     ... = ctx->p1;
16     ... = ctx->p2;
17     ctx->r1 = ...;
18     ...;
19 }
20
21 int func(int p1, int p2)
22 {
23     int v1, v2;
24     int r2;
25     func_cs_ctx_t ctx;
26
27     v1 = ...;
28
29     ctx.p1 = p1;
30
31     rcl_exec(&func_cs, &ctx, 0);
32
33     r2 = ...;
34
35     return ctx.r1 + r2;
36 }
```

Figure 3.2. Code de la Figure 3.1 modifié pour utiliser un RCL.

FIGURE 3 – Transformation d'une prise de verrou classique en RCL.

Afin de faciliter l'utilisation du RCL, nous avons implémenté le mécanisme décrit dans cette section dans une bibliothèque nommée rcllib.

2.3. Adaptation des applications patrimoniales

La majorité des applications multi-fil actuelles utilisent des verrous comme mécanisme principal de synchronisation. Il est possible de les modifier pour que celles-ci utilisent le RCL, au moins en remplacement de certains verrous critiques au niveau des performances. Le procédé de transformation d'un verrou classique vers un RCL s'avère être simple et automatisable : c'est l'objet de cette sous-section.

La Figure 3.1 présente le code d'une fonction `func` qui contient une section critique. Cette section critique utilise deux paramètres en entrée : `p1`, un paramètre reçu en argument par la fonction `func`, et `r1`, une variable locale de cette même fonction. La section critique utilise également un paramètre en sortie : la variable locale `r2`. Afin de simplifier la lecture, dans cet exemple, les variables sont toutes des entiers car leur type n'a pas d'importance dans le processus de transformation.

La Figure 3.2 illustre la transformation du code nécessaire pour l'utilisation d'un RCL. Les variables d'entrée et de sortie de la section critique sont isolées, et une structure `func_cs_ctx` contenant un emplacement mémoire pour chacune de ces variables est déclarée. Une fonction `func_cs` encapsulant la section critique est créée. Celle-ci prend en paramètre une instance de la structure `func_cs_ctx`. Le paramètre de la fonction `func_cs` est `void*` afin d'uniformiser le type des fonctions encapsulant des sections critiques, d'où le transtypage en ligne 11. Le contenu de la fonction `func_cs` est identique à celui de la section critique, à ceci près que les variables de contexte sont contenues dans la structure passée en paramètre. Enfin, dans la fonction `func` en elle-même, une instance de la structure `func_cs_ctx` est créée, et les variables qu'elle contient sont utilisées en lieu et place des variables locales correspondantes de la fonction `func`. Pour les paramètres de la fonction `func` utilisés dans la section critique, cependant, on est obligés d'effectuer une copie (ligne 29). La section critique en elle-même est remplacée par un appel à la fonction `rcl_exec` offerte par notre bibliothèque de gestion des RCL, rcllib. Cette fonction prend en paramètre l'adresse de la fonction à appeller pour exécuter la section critique (en l'oc-

curence `func_cs`), l'instance de la structure créée précédemment qui encapsule les variables d'entrée et de sortie (`ctx`) et un troisième paramètre identifiant le serveur à utiliser s'il y en a plusieurs. La fonction `rcl_exec` se charge de demander l'exécution de la section critique au serveur et d'attendre que la demande soit traitée, selon le protocole décrit dans la Section 2.2.
La transformation d'une exécution de section critique classique pour que celle-ci utilise un RCL est un processus automatisable au niveau des compilateurs : il suffit (i) d'isoler le contexte de la section critique, (ii) de créer une structure encapsulant les variables de contexte, (iii) de créer une fonction contenant le code modifié de la section critique, puis (iv) de remplacer la section critique par un appel à la fonction `rcl_exec`. L'étape (i) est facilement automatisable car l'isolation du contexte correspond à un calcul de fermeture, tâche que la majorité compilateurs sont déjà capables d'effectuer car elle est nécessaire au procédé de compilation. Quant aux étapes (ii), (iii) et (iv), il ne s'agit que d'injection et de réécriture de code : ces fonctionnalités sont déjà très largement utilisées par les compilateurs actuels lors des phases d'optimisation. La transformation automatique de verrous en RCL au niveau d'un compilateur est un objectif à long terme de ce travail de recherche.

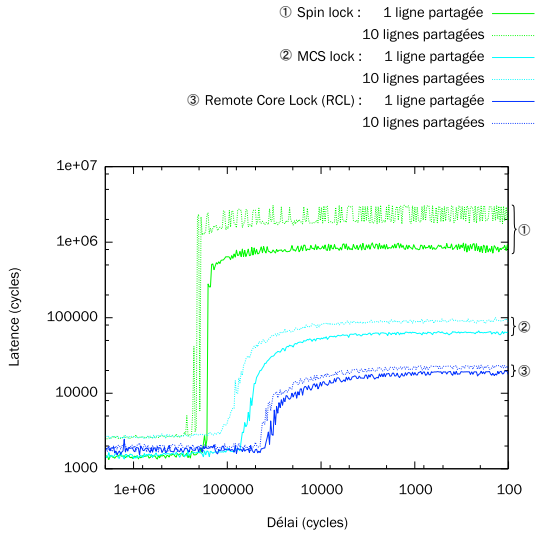
3. Évaluation

Le RCL a pour objectif d'être plus performant que les verrous classiques à la fois en ce qui concerne (i) le temps d'entrée en section critique, et ce même lorsque la contention est élevée, grâce à son mécanisme d'attente active utilisant une variable de synchronisation par client, et (ii) la durée d'exécution du contenu des sections critiques, en améliorant la localité des données partagées qui se trouvent toutes sur le cœur serveur, d'où une diminution du nombre de défauts de cache.
L'évaluation du RCL s'est déroulée en deux étapes : dans une première étape, nous avons développé un microbenchmark pour évaluer ses performances par rapport au spin lock (verrou à attente active) classique et par rapport au verrou MCS, état de l'art des spin locks actuels. Dans une deuxième étape, nous avons remplacé un verrou qui constituait un goulot d'étranglement d'abord par un spin lock, puis par un verrou MCS et enfin par un RCL dans deux applications réelles : le benchmark Raytrace de la suite SPLASH-2, et le serveur de cache memcached.
Toutes les expériences ont été réalisées sur une machine utilisant un chipset AMD RD 890 et dotée de quatre microprocesseurs Opteron « Magny-Cours » offrant chacun 12 cœurs répartis sur deux dies (48 cœurs au total). Le système d'exploitation utilisé était Mandriva 2010.1 (x64) avec le noyau Linux 2.6.33.

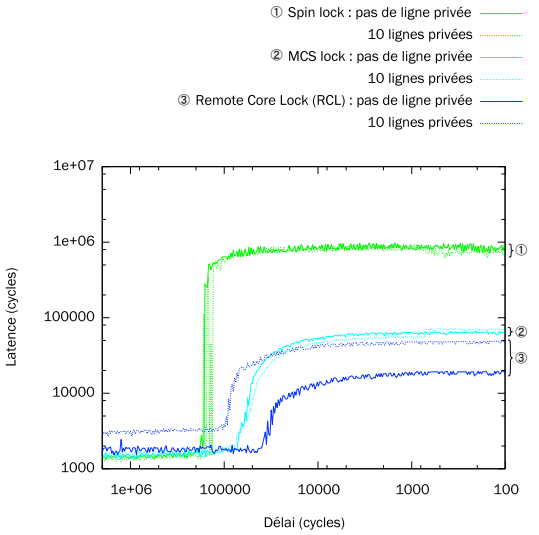
3.1. Microbenchmark

En première phase d'évaluation, il était nécessaire de disposer d'un microbenchmark permettant de faire varier le nombre de variables partagées et privées auxquelles accèdent les sections critiques, afin d'évaluer les cas dans lesquel le RCL est plus performant que les verrous classiques. Afin de satisfaire à ce besoin, très spécifique, nous avons développé notre propre microbenchmark. Celui-ci permet de comparer la latence d'exécution des sections critiques, c'est-à-dire leur durée d'exécution en incluant le temps de prise de verrou, pour (i) les spin locks classiques, (ii) le verrou MCS proposé par Scott et al. [24] et qui souvent considéré comme l'état de l'art des spin locks à l'heure actuelle, et (iii) le RCL. Nous n'avons pas mesuré les performances des verrous bloquants dans ce microbenchmark, car leur latence est trop élevée pour être comparable à celle des verrous à attente active.
Le fonctionnement du microbenchmark est le suivant : des fils de contrôle « clients » sont créés ainsi qu'un fil de contrôle serveur dans le cas où on mesure la performance du RCL. Chacun de ces fils clients et serveur sont associés à un cœur de façon à ce que tous les cœurs soient utilisés dans le cas du RCL – on a donc 47 clients et éventuellement un serveur. Un ensemble de variables sont partagées entre tous les fils de contrôle clients, et chaque client dispose d'un ensemble de variables privées. Dans chacun de ces ensembles, les variables sont espacées de manière à ce que deux d'entre elles ne correspondent jamais à la même ligne de cache au niveau du matériel, afin qu'un accès à une variable modélise un accès à une ligne de cache. Lors de l'exécution, les clients exécutent des sections critiques dans lesquelles ils accèdent à chacune des variables partagées et privées. Les clients accèdent également aux variables privées avant d'entrer en section critique afin d'assurer que celles-ci ne restent pas dans le cache du

2. On considère qu'une section critique contient toujours au moins un accès à une variable partagée, ce qui est effectivement le cas le plus courant.



Expérience a. On n'utilise pas de lignes de cache privées et le nombre de lignes partagées varie.



Expérience b. On n'accède qu'à une seule ligne de cache partagée [2], et le nombre de lignes privées varie.

FIGURE 4 – Comparaison des performances du RCL par rapport à celles du spin lock classique et du verrou MCS, mesurées à l'aide de notre microbenchmark. Accéder à 10 lignes de cache dans les sections critiques suffit à rendre le RCL plus efficace que les spin locks dans un scénario de forte contention.

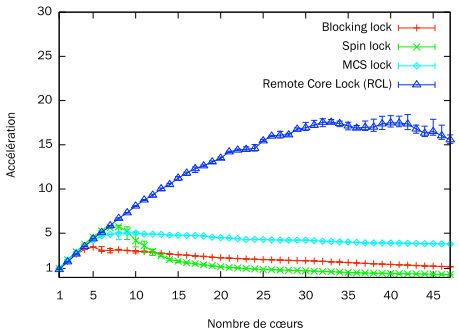
serveur et ainsi modéliser correctement le comportement de variables privées. Entre chaque exécution de sections critiques, les clients attendent activement pendant un délai donné. Il est ainsi possible de simuler différents niveau de contention : plus le délai est court, plus les prises de verrou sont fréquentes et plus la contention est élevée.

La Figure 4 présente les résultats de deux expériences effectuées à l'aide de notre microbenchmark et comparant les performances du RCL par rapport à celles du spin lock classique et du verrou MCS. Les performances du RCL ne sont pas dégradées lorsque les sections critiques accèdent à des variables partagées, de la même manière que les performances des spin locks (classique et MCS) ne sont pas dégradées lorsque les sections critiques accèdent à des variables privées. En effet, dans le cas du RCL, accéder à des variables privées engendre des défauts de cache car ces variables doivent être rapatriées sur le cœur serveur, de la même manière que dans le cas des verrous classiques, accéder à des variables partagées engendre des défauts de cache car ces variables doivent être rapatriées sur le cœur qui exécute la section critique.

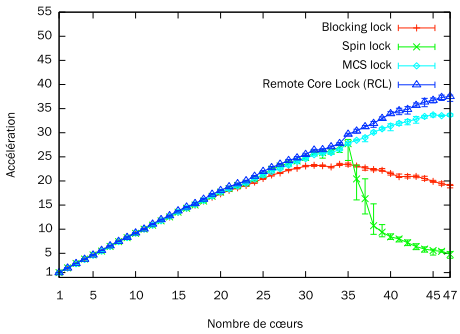
Le RCL est toujours plus efficace que les spin locks lorsque la contention est forte, et accéder à 10 lignes de cache dans les sections critiques suffit à le rendre plus efficace que les spin locks dans un scénario de faible contention. L'accès à des variables privées est certes plus coûteux pour le RCL lorsque la contention est faible, mais par leur nature, les sections critiques ont tendance à manipuler plus de données partagées que de données privées. Notons également qu'étant donné que le contexte, c'est-à-dire les variables privées, est encapsulé dans une structure (cf. Section 2), celles-ci auront tendance à se trouver isolées sur peu de lignes de cache. Ainsi, dans un scénario raisonnable dans lequel les sections critiques accèdent à des structures de données complexes dans les sections critiques et à un nombre limité variables privées servant de paramètres d'exécution, le RCL devrait s'avérer plus performant que les spin locks quel que soit le niveau de contention.

3.2. Macrobenchmarks

La seconde série d'évaluations a consisté à mesurer les performances du RCL dans deux applications réelles, afin d'estimer sa capacité à améliorer leurs performances lorsque la contention est forte d'une part, et lorsque les sections critiques accèdent à un grand nombre de variables partagées d'autre part. Les deux applications choisies pour cette évaluation sont (i) le benchmark Raytrace de la suite SPLASH-2, dont le goulot d'étranglement lorsque le nombre de cœurs d'exécution utilisés est élevé est un verrou sujet à une forte contention et (ii) le serveur de cache memcached qui utilise un seul verrou pour protéger tous les accès à sa table de hachage principale, d'où un accès à de nombreuses variables partagées dans



Expérience a. Rendu de la scène « car ».



Expérience b. Rendu de la scène « balls4 ».

FIGURE 5 – Performances du benchmark Raytrace en utilisant différents verrous pour le « rid lock ». Chaque mesure est moyénnée sur 5 exécutions. L'accélération est calculée par rapport à l'application non modifiée n'utilisant qu'un seul fil de contrôle.

les sections critiques. Ces applications utilisent les verrous bloquants de la bibliothèque pthread par défaut. Pour chacune des deux applications, le verrou qui constituait un goulot d'étranglement a tour à tour été remplacé par un spin lock, par un verrou MCS puis par un RCL.

3.2.1. SPLASH-2/Raytrace

Raytrace est un programme de ray-tracing provenant de la suite de benchmarks SPLASH-2 [3] (Stanford Parallel Applications for Shared Memory 2). SPLASH-2 date de 1994 et a pour objectif d'évaluer les performances des architectures multi-cœur. Dans Raytrace, à cause de l'évolution de la puissance des microprocesseurs, les temps de calculs sont devenus négligeables par rapport au coût des prises fréquentes du verrou « rid lock », qui est utilisé pour incrémenter le compteur de rayons : ce verrou constitue un goulot d'étranglement majeur sur notre machine de test. Les sections critiques protégées par le verrou « rid lock » n'accèdent qu'à une seule variable partagée.

La Figure 5 présente l'accélération par rapport à la version de base de Raytrace (un seul cœur, verrous bloquants) en fonction du nombre de cœurs utilisés, de quatre versions de Raytrace. Celles-ci utilisent respectivement le verrou bloquant de base, le spin lock, le verrou MCS et le Remote Core Lock (RCL). Dans tous les cas, on a autant de fils de contrôle que de cœurs et chaque fil de contrôle est lié à un cœur particulier. Les jeux de données en entrée sont les deux scènes « car » et « balls4 » qui sont fournies avec le benchmark[2]. La scène « car » engendre des niveaux de contention plus élevés que la scène « balls4 » car elle utilise une hiérachie plus profonde pour stocker les objets de la scène, d'où une diminution des temps de calcul.

Même si le verrou « rid lock » n'accède qu'à une seule variable partagée dans les sections critiques, les gains en performance et en scalabilité sont importants. Pour la scène « car », le RCL permet d'atteindre une accélération maximale de 18× à 32 cœurs, contre une accélération maximale de 6× à 8 cœurs pour le spin lock classique, une accélération maximale de 5× à 9 cœurs pour le verrou MCS, et une accélération maximale de 3× à 5 cœurs pour le verrou bloquant. Pour la scène « balls4 », le RCL permet d'atteindre une accélération maximale de 38× à 47 cœurs, contre une accélération maximale de 34× à 47 cœurs pour le verrou MCS, une accélération maximale de 28× à 35 cœurs pour le spin lock classique, et une accélération maximale de 23× à 31 cœurs pour le verrou bloquant.

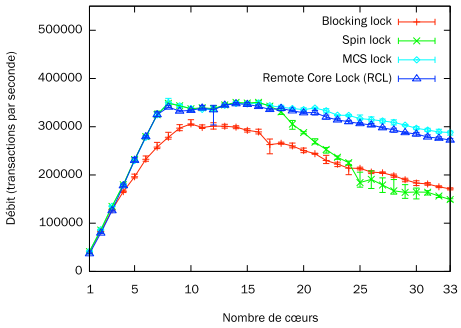
Ces résultats montrent que le RCL passe mieux à l'échelle que les verrous bloquants et à attente active, offrant des gains de performance importants lorsque le nombre de cœurs d'exécution utilisés est elevé.

3.2.2. Memcached

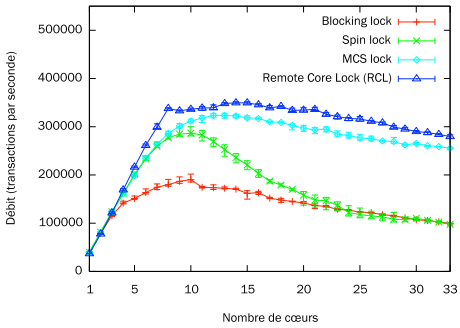
Memcached [2] est un serveur de cache populaire dans les applications web nécessitant de résister à une forte charge. Le verrou que nous avons remplacé est le « cache lock », qui protège tous les accès à la table de hachage centrale utilisée par le serveur memcached lorsque celui-ci est exécuté en mode multi-fil.

Les tests ont été effectués avec memcached 1.4.5 la bibliothèque libmemcached 0.44 [1]. Libmemcached fournit un client appelé memslap permettant de simuler une charge réaliste sur une instance du serveur

2. Nous n'avons pas présenté les résultats pour la troisième scène, « teapot », car les auteurs de SPLASH-2 expliquent qu'il ne s'agit que d'une scène extrêmement basique n'ayant pour objectif que de vérifier le bon fonctionnement de l'application.



Expérience a. 10% de requêtes set.



Expérience b. 30% de requêtes set.

FIGURE 6 – Performances de memcached, en utilisant différents verrous pour le « cache lock ». Chaque mesure est moyénnée sur 10 exécutions.

memcached. Dans nos expériences, memslap et memcached s'exécutent en même temps sur notre machine de test, 12 cœurs étant réservés pour l'exécution de memslap et les cœurs restants étant dédiés à memcached : cette configuration suffit pour saturer memcached jusqu'à 36 cœurs, quel que soient les verrous utilisé. Memcached est lancé en mode multi-fils avec les paramètres par défaut. Le nombre de fils de contrôle de travail de memcached varie de 1 à 33, car un cœur est réservé pour le serveur RCL et deux autres cœurs sont réservés à deux fils de contrôles spécifiques à memcached traitant respectivement (i) les requêtes entrantes et (ii) les opérations de maintenance de la table de hachage (augmentation ou diminution de sa taille, par exemple).

Deux expériences sont présentées sur la Figure 6. Dans l'Expérience 6.a, le taux de requêtes « set » permettant de modifier le contenu de la table de hache principale est faible (10%). Cette expérience mesure donc un cas où la table de hachage est principalement accédée en lecture. Nos résultats montrent que le RCL permet d'obtenir des performances similaires au spin lock et au verrou MCS dans ce cas. Cette expérience n'est cependant pas favorable au RCL, car celui-ci ne devient efficace que lorsque les sections critiques accèdent à de nombreuses données partagées, et la lecture des données dans une table de hachage ne nécessite d'accéder qu'à un seul emplacement mémoire le plus souvent.

Dans l'Expérience 6.b, le taux de requêtes set est plus élevé (30%). Dans ce cas, memcached offre des gains de débits de 6%, 16% et de 65% respectivement par rapport au verrou MCS, au spin lock et au verrou bloquant respectivement, car les opérations d'écriture dans la table de hachage nécessitent l'accès à de nombreuses données partagées.

Nos mesures montrent donc que le RCL permet d'obtenir des performances similaires au spin lock et au verrou MCS dans la première expérience, et permet d'obtenir des gains de performance non négligeables par rapport à ces verrous dans la seconde expérience. Les accès en écriture à la base (requêtes « set ») accédant à beaucoup plus de lignes de cache que les accès en lecture, ces gains sont bien dûs à la localité des données.

Que ce soit dans l'Expérience 6.a ou dans l'Expérience 6.b, les performances de memcached atteignent un plateau à partir de 12 cœurs, que ce soit avec le RCL, le spin lock ou les verrous bloquants. Cette stagnation des performances est probablement due à un goulot d'étranglement indépendant des prises de verrou que nous sommes actuellement en train de chercher à identifier.

Cette première série d'expériences montre que le RCL permet de mieux passer à l'échelle dans les deux cas suivants : (i) lorsque de nombreux fils de contrôle essaient d'exécuter des sections critiques de façon concurrente d'une part, grâce à l'implémentation efficace du RCL qui utilise une variable de synchronisation par client, et (ii) lorsque les sections critiques accèdent à de nombreuses données partagées d'autre part, grâce au gain de localité engendré par le RCL.

4. État de l'art

Pour améliorer les performances des applications multi-fils à mémoire partagée sur les architectures multi-cœur, deux pistes principales ont été explorées : (i) l'amélioration des fonctions de verrouillage et (ii) les mémoires transactionnelles.

Amélioration des algorithmes de gestion des verrous

Une première méthode permettant d'améliorer les algorithmes de gestion des verrous est l'amélioration des fonctions de verrouillage. Trois principales séries de travaux ont proposé des approches différentes pour traiter ce problème. La première série de travaux [5, 28, 31] vise à supprimer les prises inutiles de verrous lorsqu'une donnée n'est accédée que par une unique tâche. La seconde série [29, 8, 23, 11, 4, 22] vise à améliorer l'équité des algorithmes ou à diminuer la charge sur le bus lors d'une prise de verrou. La troisième série [19, 33] vise quant à elle à choisir le meilleur algorithme entre l'attente passive et l'attente active en fonction de la contention observée sur le verrou. Contrairement au RCL, aucune de ces trois optimisations ne prend en compte la problématique de la localité des données lors de l'exécution des sections critiques : en effet, un verrou sera maintenu plus longtemps pour l'exécution d'une section critique si les données partagées auxquelles elle accède ne sont pas présentes dans ses caches locaux, voire dans ses mémoires locales dans le cas d'une architecture NUMA.
Une seconde méthode permettant d'améliorer les algorithmes de verrouillage est l'utilisation de modèles de synchronisation spécifiques comme par exemple le modèle de verrouillage lecteurs-écrivains qui permet à plusieurs fils de contrôle lecteurs d'exécuter une section critique en parallèle du moment qu'aucun fil de contrôle écrivain ne possède le verrou, ou le modèle « read copy update » [12] (implémenté dans le noyau Linux [20]) qui ne bloque jamais les fils de contrôle lecteurs au prix d'un niveau de performances dégradé pour les fils de contrôle écrivains. Ces techniques sont utiles si les motifs d'accès aux variables partagées sont prévisibles, mais ne sont pas applicables dans le cas général. Notons également qu'encore une fois, contrairement au RCL, aucune de ces techniques ne prend en compte la localité des données pour les sections critiques.
De nombreux algorithmes ont également été proposés afin d'éviter l'utilisation de verrous pour des structures de données classiques telles que des compteurs, listes chaînées, files, piles ou tables de hachage [18, 30, 21, 16, 26, 17, 7]. Ces algorithmes sont dits non-bloquants car ils ne bloquent jamais un fil de contrôle lors d'un accès – en lecture ou en écriture – à la structure de données concernée. Ils sont donc très efficaces sur les architectures multi-cœur. Cependant, ces algorithmes ne permettent d'effectuer qu'un nombre limité de manipulations sur des structures de données. Par exemple, un algorithme proposé par Scott [26] permet d'ajouter ou de supprimer un élément d'une liste FIFO, mais ne peut être utilisé s'il est parfois nécessaire de déplacer un élément d'une file à une autre de manière atomique. C'est pourquoi les algorithmes basés sur des verrous restent le standard utilisé sur les architectures multi-cœur. Le travail d'amélioration des techniques de verrouillages, auquel nous contribuons avec le RCL, reste donc crucial à l'heure actuelle.

Utilisation de mémoires transactionnelles

Les mémoires transactionnelles [13, 15, 25] ont été étudiées durant les dix dernières années dans l'objectif de faciliter le développement des applications multi-fil. Elles permettent l'exécution optimiste de blocs de code marqués « atomiques » de manière à ce que les écritures d'un fil de contrôle semblent avoir été exécutées atomiquement du point de vue des autres fils de contôle. L'utilisation de mémoires transactionnelles simplifie le développement d'applications multi-fil, en particulier parce qu'elles permettent de composer les blocs atomiques entre eux et parce leur utilisation rend la présence d'interblocages impossible [14].
Une méthode permettant de transformer de manière automatique les applications patrimoniales Java pour qu'elle utilisent les mémoires transactionnelles a été proposé par Ziarek et al. [32]. Cette transformation est réalisée par la machine virtuelle Java, qui décide à l'exécution si un bloc synchronisé doit être ou non implémenté en tant que bloc atomique. Cependant, l'utilisation de mémoires transactionnelles nécessite d'instrumenter les lectures et les écritures pour annuler l'exécution d'une transaction via un rollback lorsqu'un conflit est détecté. Cette instrumentation est coûteuse même lorsqu'elle est gérée par le matériel [27]. Le RCL ne nécessite pas de technique coûteuse à l'exécution comme cette instrumentation des accès à la mémoire et permet en général d'obtenir des performances similaires ou meilleures que les implémentations classiques.

5. Conclusion

Cet article présente une nouvelle technique de synchronisation appelée le Remote Core Lock (RCL) qui centralise sur un serveur unique l'exécution de sections critiques. L'avantage du RCL est double : d'une part, grâce à son architecture client/serveur et à l'utilisation d'une variable de synchronisation par client, le RCL permet de diminuer le temps d'entrée en section critique lorsque de nombreux clients essaient d'exécuter des sections critiques de façon concurrente, et d'autre part, grâce à une meilleure localité des données, la durée d'exécution des sections critiques est réduite.
L'évaluation du RCL montre que (i) le RCL offre des performances supérieures aux verrous classiques en cas de forte contention sur le bus, (ii) grâce au RCL, le benchmark SPLASH-2/Raytrace passe à l'échelle jusqu'à 32 cœurs, au lieu de 8 avec des verrous classiques lorsque la contention est très forte (scène « car ») et (iii) l'utilisation du RCL dans le serveur de cache memcached offre un gain de débit allant jusqu'à 65%. Dans toutes nos expériences, le RCL est également au moins aussi performant que le MCS qui est pourtant l'un des verrous à attente active les plus performants à l'heure actuelle.

Perspectives

Dans le contexte de ce travail de recherche, il est maintenant nécessaire d'affiner l'évaluation du RCL. Il faudra (i) implémenter le RCL dans chacun des benchmarks de la suite SPLASH-2 pour des raisons d'exhaustivité, (ii) implémenter le verrou MCS dans les macrobenchmarks, et (iii) isoler le goulot d'étranglement observé dans memcached (cf. Section 3.2.2). L'objectif suivant est d'isoler davantage d'applications dans lesquelles le RCL est rentable. Pour cela, il sera nécessaire de mettre au point des techniques d'instrumentation des applications patrimoniales permettant d'isoler les verrous qu'il serait rentable de remplacer par des RCL. Enfin, un objectif à plus long terme est d'étudier les possibilités de remplacement automatique de verrous par des RCL au niveau d'un compilateur.

Ce travail ouvre une nouvelle voie dans la recherche de méthodes de synchronisation efficaces pour les nouvelles architectures multi-cœur. En effet, le RCL est un verrou unique en cela qu'il a pour objectif de réduire la contention non seulement en limitant la charge sur le bus, mais aussi en réduisant la durée d'exécution des sections critiques grâce à une meilleure localité des données. Il s'agit là d'une stratégie novatrice à l'heure actuelle, où il devient crucial de trouver des solutions permettant de compenser le coût prohibitif des algorithmes de cohérence de cache sur les architectures à grand nombre de cœurs.

Bibliographie

1. Libmemcached. `https://launchpad.net/libmemcached`.
2. Memcached. `http://memcached.org`.
3. SPLASH-2. `http://www.capsl.udel.edu/splash`.
4. T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. In *Transactions on Parallel and Distributed Systems*, volume 1, pages 6–16. IEEE Computer Society Press, January 1990.
5. D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 258–268, New York, NY, USA. ACM Press.
6. A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM symposium on Operating systems principles*, SOSP '09, pages 29–44, Big Sky, MO, USA, October 2009. ACM Press.
7. S. Boyd-Wickizer, A. T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, Vancouver, Canada, October 2010. USENIX Association.
8. T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, February 2003.
9. F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 186–189, Saint-Émilion, France, 2002. ACM Press.
10. Fabien Gaud, Sylvain Geneves, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma. Efficient Workstealing for Multicore Event-Driven Systems. In *Proceedings of the International Conference on Distributed Computing Systems*, ICDCS '10, Genoa, Italy, June 2010. IEEE Computer Society Press.

11. G. Granunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *Computer*, 23(6):21–65, June 1990.
12. D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, April 2008.
13. T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27:8–29, 2007.
14. T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoP '05, pages 48–60, Chicago, IL, USA, June 2005. ACM Press.
15. T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '03, pages 388–402, Anaheim, CA, USA, October 2003.
16. D. Hendler, N. Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 206–215, New York, NY, USA. ACM Press.
17. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
18. M. Herlihy and N. Shavit. *The Art Of Multiprocessor Programming*. Elsevier, 2008.
19. F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *Proceedings of the 15th edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 117–128, New York, NY, USA. ACM Press.
20. P. Mc Kenny. Introduction to RCU. `http://www.rdrop.com/users/paulmck/RCU/`.
21. E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. *Distributed Computing*, 20(5):323–341, 2008.
22. P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, IPPS '94, pages 165–171, Cancun, Mexico, April. IEEE Computer Society Press.
23. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
24. John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 269–278, 1991.
25. V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 314–325, Munich, Germany, 2008. ACM Press.
26. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA. ACM Press.
27. Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 87–102, New York, NY, USA. ACM Press.
28. K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 263–272, New York, NY, USA. ACM Press.
29. M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *Scalable queue-based spin locks with timeout*, PPoPP '01, pages 44–52, New York, NY, USA. ACM Press.
30. O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, May 2006.
31. N. Vasudevan, K. S. Namjoshi, and Stephen A. Edwards. Simple and fast biased locks. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 65–74, New York, NY, USA. ACM Press.
32. L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for Java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 129–154, Paphos, Cyprus, July 2008. Springer-Verlag.
33. P. Zijlstra. Mutex: implement adaptive spinning, January 2009. E-mail to Linus Torvalds on LWN.net (`http://lwn.net/Articles/314512/`).